cepton_sdk Documentation

Cepton Technologies

Jul 10, 2019

CONTENTS

CHAPTER

ONE

OVERVIEW

The Cepton SDK provides the following features

- **Parsing**: parse sensor packets
- **Calibration**: apply sensor calibration
- **Networking**: listen for sensor packets
- **Capture Replay**: read sensor packets from a PCAP file
- **Frame Accumulation**: accumulate and output sensor points by frame

Currently, the Cepton LiDAR packet formats are under active development, and are not publicly available. The SDK is required for **Parsing** and **Calibration**. All other SDK features are optional, and can be done manually by the user.

For prototyping, it is recommended to start with the high level methods in *API* and *Utilities*.

1.1 Timestamps

All `int64` timestamps are microseconds since the Unix epoch (UTC). Point timestamps are based on one of the following sources (the first valid source is used):

1. GPS (NMEA + PPS)
2. PTP
3. Host PC

All `float` and `int64` time differences are seconds (measurement period, replay time, frame length, etc.).

1.2 Multiple Returns

To enable multiple returns, pass the `CEPTON_SDK_CONTROL_ENABLE_MULTIPLE_RETURNS` flag during initialization.

The returns are as follows:

1. Strongest signal.
2. Furthest signal, if it is not the strongest. Otherwise, the second strongest signal.

1.3 Concurrency

- If networking is enabled, the sdk creates 2 threads for networking. All callbacks occur in these networking threads.
- If networking is disabled, the sdk does not create any threads. All callbacks occur in the main thread.
- All sdk getter functions are thread safe, and can be called from callbacks. Other sdk functions are not guarunteed to be thread safe, and can cause deadlock if called from callbacks.

1.4 Minimal SDK

If desired, the following SDK features can be disabled in the SDK and performed manually by the user:

- Networking: *Network*.
- Capture Replay: *Replay*.
- Frame Accumulation: *Frame*.

CHAPTER

TWO

ERRORS

2.1 Types

class SensorError : **public** runtime_error

Error returned by most functions.

Implicitly convertible from/to SensorErrorCode. Getter functions do not return an error, because they cannot fail. Will call `CEPTON_RUNTIME_ASSERT` if nonzero error is not checked (call `ignore` to manually use error).

Public Functions

SensorError(*SensorErrorCode* *code_*, **const** std::string &*msg_*)

SensorError(*SensorErrorCode* *code_*)

SensorError()

~SensorError()

SensorError(**const** *SensorError* &*other*)

SensorError &**operator=**(**const** *SensorError* &*other*)

bool **used**() **const**

Internal use only.

void **ignore**() **const**

Mark error as used.

const std::string &**msg**() **const**

Returns error message;.

SensorErrorCode **code**() **const**

Returns error code.

operator SensorErrorCode() **const**

Implicitly convert to `SensorErrorCode`.

operator bool() **const**

Returns `false` if code is `CEPTON_SUCCESS`, true otherwise.

const char ***name**() **const**

bool **is_error**() **const**

bool **is_fault**() **const**

_CeptonSensorErrorCode

Values:

0

1

2

4

5

6

Networking error.

7

8

Invalid value or uninitialized struct.

9

10

11

12

13

14

15

1000

Internal parameter out of range.

1001

Reading exceed spec.

1002

Reading exceeds spec.

1003

1004

1005

1006

1007

1008

typedef CeptonSensorErrorCode cepton_sdk**::SensorErrorCode**

typedef int32_t **CeptonSensorErrorCode**

2.2 Methods

const char *cepton_sdk::**get_error_code_name**(*SensorErrorCode error_code*)
Returns string name of error code.

Returns empty string if error code is invalid.

bool cepton_sdk::**is_error_code**(*SensorErrorCode error_code*)
Returns true if error name is of the form CEPTON_ERROR_*, false otherwise.

bool cepton_sdk::**is_fault_code**(*SensorErrorCode error_code*)
Returns true if error name is of the form CEPTON_FAULT_*, false otherwise.

SensorError cepton_sdk::**get_error**()
Returns and clears the last sdk error.

Called automatically by all C++ methods, so only useful when calling C methods directly.

CHAPTER

THREE

SETUP

3.1 Types

_CeptonSDKControl
SDK control flags.

Values:

1
Disable networking operations.

Useful for running multiple instances of sdk in different processes. Must pass packets manually to `cepton_sdk::mock_network_receive`.

2
Disable marking image clipped points as invalid.

Does not affect number of points returned.

3
Disable marking distance clipped points as invalid.

Does not affect number of points returned.

4
Enable multiple returns.

When set, *`cepton_sdk::SensorInformation::return_count`* will indicate the number of returns per laser. Can only be set at sdk initialization.

5
Enable marking stray points as invalid (measurement noise).

Uses `cepton_sdk::util::StrayFilter` to mark points invalid.

Does not affect number of points returned.

6
Always use packet timestamps (disable GPS/PTP timestamps).

7
Enable marking crosstalk points as invalid.

typedef CeptonSDKControl `cepton_sdk::`**Control**

typedef uint32_t **CeptonSDKControl**

_CeptonSDKFrameMode
Controls frequency of points being reported.

Values:

0
Report points by packet.

1
Report points at fixed time intervals.

Interval controlled by *CeptonSDKFrameOptions::length*.

2
Report points when the field of view is covered once.

- For HR80 series, detects half scan cycle (left-to-right or right-to-left).

3
Report points when the scan pattern goes through a full cycle.

Typically 2x longer frame than COVER mode.

- For HR80 series, detects full scan cycle (left-to-right-to-left).
- For VISTA series, internally uses TIMED mode.

3

typedef CeptonSDKFrameMode `cepton_sdk::`**FrameMode**

typedef uint32_t **CeptonSDKFrameMode**

struct CeptonSDKFrameOptions

Public Members

size_t **signature**
Internal use only.

CeptonSDKFrameMode **mode**
Default: CEPTON_SDK_FRAME_STREAMING.

float **length**
Frame length [seconds].

Default: 0.05. Only used if mode=CEPTON_SDK_FRAME_TIMED.

typedef CeptonSDKFrameOptions `cepton_sdk::`**FrameOptions**

FrameOptions `cepton_sdk::`**create_frame_options**`()`
Create default frame options.

struct CeptonSDKOptions
SDK initialization options.

Public Members

size_t **signature**
Internal use only.

CeptonSDKControl **control_flags**
Default: 0.

struct *CeptonSDKFrameOptions* **frame**

uint16_t **port**
Default: 8808.

typedef CeptonSDKOptions cepton_sdk::**Options**

Options cepton_sdk::**create_options**()
Create default options.

typedef void (*cepton_sdk::**FpSensorErrorCallback**)(*SensorHandle* handle, *SensorErrorCode* error_code, **const** char *error_msg, **const** void *error_data, size_t error_data_size, void *user_data)
Callback for receiving sdk and sensor errors.

Currently, error_data is not used.

3.2 Methods

SensorError cepton_sdk::**initialize**(int *version*, **const** *Options* &*options* = *create_options*(), **const** *FpSensorErrorCallback* &*cb* = nullptr, void ***const** *user_data* = nullptr)
Initializes settings and networking.

Must be called before any other sdk function listed below.

SensorError cepton_sdk::**deinitialize**()
Resets everything and deallocates memory.

SensorError cepton_sdk::**set_control_flags**(*Control mask*, *Control flags*)

Control cepton_sdk::**get_control_flags**()

bool cepton_sdk::**has_control_flag**(*Control flag*)

uint16_t cepton_sdk::**get_port**()

SensorError cepton_sdk::**set_port**(uint16_t *port*)
Sets network listen port.

Default: 8808.

SensorError cepton_sdk::**set_frame_options**(**const** CeptonSDKFrameOptions &*options*)

FrameMode cepton_sdk::**get_frame_mode**()

float cepton_sdk::**get_frame_length**()

CHAPTER

FOUR

SENSORS

4.1 Types

typedef CeptonSensorHandle `cepton_sdk::`**SensorHandle**
Sensor identifier.

_CeptonSensorModel
Values:

1

2

3

4

5

6

7

8

9

10

11

12

12

typedef CeptonSensorModel `cepton_sdk::`**SensorModel**

typedef CeptonSensorInformation `cepton_sdk::`**SensorInformation**

struct CeptonSensorInformation

Public Members

CeptonSensorHandle **handle**

uint64_t **serial_number**

char CeptonSensorInformation::model_name[28]

CeptonSensorModel **model**

uint16_t **reserved**

char CeptonSensorInformation::firmware_version[28]

uint8_t **major**

uint8_t **minor**

uint8_t CeptonSensorInformation::unused[2]

struct *CeptonSensorInformation*::@16 **formal_firmware_version**

float **last_reported_temperature**
[celsius]

float **last_reported_humidity**
[%]

float **last_reported_age**
[hours]

float **measurement_period**
Time between measurements [seconds].

int64_t **ptp_ts**
[microseconds]

uint8_t **gps_ts_year**
0-99 (2017 -> 17)

uint8_t **gps_ts_month**
1-12

uint8_t **gps_ts_day**
1-31

uint8_t **gps_ts_hour**
0-23

uint8_t **gps_ts_min**
0-59

uint8_t **gps_ts_sec**
0-59

uint8_t **return_count**

uint8_t **segment_count**
Number of image segments.

uint32_t **flags**

uint32_t **is_mocked**
Created by capture replay.

uint32_t **is_pps_connected**
GPS PPS is available.

uint32_t **is_nmea_connected**
GPS NMEA is available.

uint32_t **is_ptp_connected**
PTP is available.

uint32_t **is_calibrated**

uint32_t **is_over_heated**
Hit temperature limit (only available in Vista Gen2 for now)

uint32_t **is_sync_firing_enabled**

union *CeptonSensorInformation*::@17 *CeptonSensorInformation*::**@18**

4.2 Methods

std::size_t cepton_sdk::**get_n_sensors**()
Get number of sensors attached. Use to check for new sensors. Sensors are not deleted until deinitialization.

SensorError cepton_sdk::**get_sensor_handle_by_serial_number**(uint64_t *serial_number*, *SensorHandle* &*handle*)
Returns error if sensor not found.

SensorError cepton_sdk::**get_sensor_information_by_index**(std::size_t *idx*, *SensorInformation* &*info*)
Valid indices are in range [0, n_sensors). Returns error if index invalid.

SensorError cepton_sdk::**get_sensor_information**(*SensorHandle* *handle*, *SensorInformation* &*info*)
Returns error if sensor not found.

CHAPTER

FIVE

POINTS

5.1 Types

typedef CeptonSensorImagePoint `cepton_sdk::SensorImagePoint`

struct CeptonSensorImagePoint

Point in image coordinates (focal length = 1).

To convert to 3d point, refer to `cepton_sdk_util.hpp`.

Public Members

int64_t **timestamp**

Unix time [microseconds].

float **image_x**

x image coordinate.

float **distance**

Distance [meters].

float **image_z**

z image coordinate.

float **intensity**

Diffuse reflectance.

CeptonSensorReturnType **return_type**

uint8_t **flags**

uint8_t **valid**

uint8_t **saturated**

union *CeptonSensorImagePoint*::@21 *CeptonSensorImagePoint*::**@22**

uint8_t CeptonSensorImagePoint::reserved[2]

5.2 Methods

typedef void **(* FpCeptonSensorImageDataCallback)** (CeptonSensorHandle *handle*, size_t *n_points*, const struct *CeptonSensorImagePoint* *c_points*, void *user_data*)

SensorError cepton_sdk::**listen_image_frames**(FpSensorImageDataCallback *cb*, void *const *user_data*)

Sets image frames callback.

Returns points at frequency specified by *cepton_sdk::FrameOptions::mode*. Each frame contains all possible points (use *cepton_sdk::SensorImagePoint::valid* to filter points). Points are ordered by measurement, segment, and return:

```
measurement_count = n_points / (segment_count * return_count)
idx = ((i_measurement) * segment_count + i_segment) * return_count + i_return
```

Returns error if callback already registered.

SensorError cepton_sdk::**unlisten_image_frames**()

CHAPTER

SIX

NETWORKING

6.1 Types

typedef void (*cepton_sdk::**FpNetworkReceiveCallback**)(*SensorHandle* handle, int64_t timestamp, uint8_t **const** *buffer, size_t buffer_size, void *user_data)

Callback for receiving network packets.

Parameters

- handle: Unique sensor identifier (e.g. IP address). Returns error if callback already set.

6.2 Methods

SensorError cepton_sdk::**listen_network_packets**(*FpNetworkReceiveCallback* *cb*, void ***const** *user_data*)

Sets network packets callback.

Only 1 callback can be registered.

SensorError cepton_sdk::**unlisten_network_packets**()

SensorError cepton_sdk::**clear**()

Clears sensors.

Use when loading/unloading capture file.

CHAPTER

SEVEN

CAPTURE REPLAY

PCAP capture file replay. Source code can be found in the *source* folder. Functions are not thread safe, and should only be called from the main thread.

bool cepton_sdk::capture_replay::**is_open**()

SensorError cepton_sdk::capture_replay::**open**(**const** std::string &*path*)
Opens capture file.

Must be called before any other replay functions listed below.

SensorError cepton_sdk::capture_replay::**close**()

const char *cepton_sdk::capture_replay::**get_filename**()

uint64_t cepton_sdk::capture_replay::**get_start_time**()
Returns capture start timestamp (unix time [microseconds]).

float cepton_sdk::capture_replay::**get_position**()
Returns capture file position [seconds].

float cepton_sdk::capture_replay::**get_length**()
Returns capture file length [seconds].

bool cepton_sdk::capture_replay::**is_end**()
Returns true if at end of capture file.

This is only relevant when using `resume_blocking` methods.

SensorError cepton_sdk::capture_replay::**seek**(float *position*)
Seek to capture file position [seconds].

Position must be in range [0.0, capture length). Returns error if position is invalid.

SensorError cepton_sdk::capture_replay::**set_enable_loop**(bool *value*)
If enabled, replay will automatically rewind at end.

bool cepton_sdk::capture_replay::**get_enable_loop**()

SensorError cepton_sdk::capture_replay::**set_speed**(float *speed*)
Replay speed multiplier for asynchronous replay.

float cepton_sdk::capture_replay::**get_speed**()

SensorError cepton_sdk::capture_replay::**resume_blocking_once**()
Replay next packet in current thread without sleeping.

Pauses replay thread if it is running.

SensorError cepton_sdk::capture_replay::**resume_blocking**(float *duration*)
Replay multiple packets synchronously.

No sleep between packets. Resume duration must be non-negative. Pauses replay thread if it is running.

bool cepton_sdk::capture_replay::**is_running**()
Returns true if replay thread is running.

SensorError cepton_sdk::capture_replay::**resume**()
Packets are replayed in realtime. Replay thread sleeps in between packets.

SensorError cepton_sdk::capture_replay::**pause**()
Pauses asynchronous replay thread.

CHAPTER

EIGHT

API

High level SDK api for prototyping (*cepton_sdk_api.hpp*). Methods are agnostic to live/replay mode.

bool `cepton_sdk::api::`**`is_live`**`()`
Returns true if capture replay is not open.

bool `cepton_sdk::api::`**`is_end`**`()`

int64_t `cepton_sdk::api::`**`get_time`**`()`
Returns capture replay time or live time.

SensorError `cepton_sdk::api::`**`wait`**`(`float *t_length* = -1.0f`)`
Sleeps or resumes capture replay for duration.

If `t_length < 0`, then waits forever.

8.1 Errors

`const` *SensorError* &`cepton_sdk::api::`**`check_error`**`(`**`const`** *SensorError* *&error*, **`const`** std::string *&msg* = "")
Handles error.

If error, raises exception. Otherwise, prints error.

`const` *SensorError* &`cepton_sdk::api::`**`log_error`**`(`**`const`** *SensorError* *&error*, **`const`** std::string *&msg* = "")
Prints error.

void `cepton_sdk::api::`**`default_on_error`**`(`*SensorHandle h*, *SensorErrorCode error_code*, **`const`** char ***`const`** *error_msg*, **`const`** void ***`const`** *error_data*, std::size_t *error_data_size*, void ***`const`** *instance*`)`
Basic SDK error callback.

Calls `cepton_sdk::api::check_error_code`.

8.2 Setup

SensorError `cepton_sdk::api::`**`initialize`**`(`*Options options* = *create_options*(), **`const`** std::string *&capture_path* = "")
Initialize SDK and optionally starts capture replay.

`class SensorErrorCallback` : **`public`** cepton_sdk::util::Callback<*SensorHandle*, **`const`** *SensorError*&>
Callback for sensor errors.

Public Static Functions

static void **global_on_callback**(*SensorHandle* *handle*, *SensorErrorCode* *error_code*, **const** char ***error_msg*, **const** void ***const** *error_data*, size_t *error_data_size*, void ***const** *instance*)

class SensorImageFrameCallback : **public** cepton_sdk::util::Callback<*SensorHandle*, std::size_t, **const** *SensorImagePoi*

Callback for image frames.

Must call `initialize` before use.

Public Functions

~SensorImageFrameCallback()

SensorError **initialize**()

SensorError **deinitialize**()

bool **is_initialized**() **const**

class NetworkPacketCallback : **public** cepton_sdk::util::Callback<*SensorHandle*, int64_t, uint8_t **const** *, std::size_t>

Callback for network packets.

Must call `initialize` before use.

Public Functions

~NetworkPacketCallback()

SensorError **initialize**()

SensorError **deinitialize**()

8.3 Sensors

bool cepton_sdk::api::**has_sensor_by_serial_number**(uint64_t *serial_number*)

SensorError cepton_sdk::api::**get_sensor_information_by_serial_number**(uint64_t *serial_number*, *SensorInformation* &*info*)

Returns error if sensor not found.

std::vector<uint64_t> cepton_sdk::api::**get_sensor_serial_numbers**()

Returns serial numbers for all sensors.

CHAPTER

NINE

UTILITIES

Utility functions and classes for prototyping (*cepton_sdk_util.hpp*).

9.1 Common

int64_t `cepton_sdk::util::`**`get_timestamp_usec`**`()`
Returns current unix timestamp [microseconds].

This is the timestamp format used by all sdk functions.

9.2 Points

void `cepton_sdk::util::`**`convert_image_point_to_point`**(float *image_x*, float *image_z*, float *distance*, float &*x*, float &*y*, float &*z*)
Convert image point to 3d point.

`struct SensorPoint`
3d point class.

Can't subclass from SensorImagePoint, needs to be POD.

Public Members

int64_t **`timestamp`**
Unix time [microseconds].

float **`image_x`**
x image coordinate.

float **`distance`**
Distance [meters].

float **`image_z`**
z image coordinate.

float **`intensity`**
Diffuse reflectance.

CeptonSensorReturnType **`return_type`**

uint8_t **`flags`**

uint8_t **valid**

uint8_t **saturated**

union cepton_sdk::util::*SensorPoint*::**[anonymous] [anonymous]**

uint8_t **reserved**[5]

float **x**
x cartesian coordinate

float **y**
y cartesian coordinate

float **z**
z cartesian coordinate

void `cepton_sdk::util::`**convert_sensor_image_point_to_point**(**const** *SensorImagePoint* &*image_point*, *SensorPoint* &*point*)

Convenience method to convert *`cepton_sdk::SensorImagePoint`* to `cepton_sdk::SensorPoint`.

9.3 Transforms

class CompiledTransform
3d translation and rotation.

For more functionality, use Eigen's Geometry module.

Public Functions

void **apply**(float &*x*, float &*y*, float &*z*)
Apply transformation to 3d position.

Public Static Functions

static *CompiledTransform* **create**(**const** float *__const__ *translation*, **const** float *__const__ *rotation*)
Create from translation and rotation.

Parameters

- `translation`: Cartesian (x, y, z)
- `rotation`: Quaternion (x, y, z, w)

9.4 Callbacks

template<typename ...**TArgs**>
class Callback
Expands SDK callback functionality.

Allows for multiple callbacks to be registered. Allows for registering lambdas and member functions. See `samples/basic.cpp`.

Public Functions

void **clear**()
Clear all listeners.

SensorError **listen**(**const** std::function<void(TArgs...)> &*func*, uint64_t ***const*** *id* = nullptr)
Register std::function.

template<typename **TClass**>
SensorError **listen**(*TClass* ***const*** *instance*, MemberFunction<*TClass*, TArgs...> *func*, uint64_t ***const*** *id* = nullptr)
Register instance member function.

void **emit**(TArgs... *args*) **const**
Emit callback.

void **operator()**(TArgs... *args*) **const**
Emit callback.

Public Static Functions

static void **global_on_callback**(TArgs... *args*, void ***const*** *instance*)
Used for registering as c callback.

9.5 Frames

class FrameDetector
Detects frames in streaming sensor data.

Public Functions

FrameDetector(**const** *SensorInformation* &*sensor_info*)

const *FrameOptions* &**get_options**() **const**

SensorError **set_options**(**const** *FrameOptions* &*options*)

void **reset**()
Completely resets detector.

Only use if also clearing points accumulator.

bool **add_point**(**const** *SensorImagePoint* &*point*)
Returns true if frame found.

Automatically resets after frame is found.

Public Members

bool **frame_found**

int **frame_idx**

float **frame_x**
Number of points in current frame.

class FrameAccumulator
Accumulates image points, and emits frames to callback.

See `samples/frame.cpp`.

Public Functions

FrameAccumulator(**const** *SensorInformation* &*sensor_info*)

FrameOptions **get_options**() **const**

SensorError **set_options**(**const** *FrameOptions* &*options*)

void **clear**()

void **add_points**(int *n_points*, **const** *SensorImagePoint* ***const** *image_points*)

Public Members

Callback<int, **const** *SensorImagePoint* *> **callback**

9.6 Organizer

struct OrganizedCloud
The *OrganizedCloud* struct An organized version of the cepton point cloud.

Public Functions

int **getIndex**(int *row*, int *col*, int *n_return*)
getIndex Returns the index of the point corresponding to the inputed row, col and return number.

Return

Parameters

- `[in] row`: Row index
- `[in] col`: Col index
- `[in] n_return`: Return index

Public Members

int64_t **timestamp_start**
timestamp_start The time of the oldest point in the cloud

int64_t **timestamp_end**
timestamp_end The time of the newest point in the cloud

int **height**
height Height of the cloud. Represents how many rows there are in the cloud

int **width**
width Width of the cloud. Represents how many columns there are in the cloud

int **n_returns**
n_returns Number of return represented by the cloud.

std::vector<*CellInfo*> **info_cells**
info_cells Vector of cell info which provide information about the matching points

std::vector<CeptonSensorImagePoint> **points**
points Vector of organized points. Stored in Return, Row, Col order. So to get a point at row 10, col 15, return 1 would be points[(row * width

- col) n_returns + return

struct CellInfo
The *CellInfo* struct.

Public Members

bool **occupied_cell** = false
occupied_cell Is the cell at this index occupied with a point. If false can't assume this represents free space.

int **original_index** = -1
original_index Index of the point that was used to generate the organized point. Can be used to match back with orginial data if required. Should only be use if occupied_cell is true.

class Organizer
The *Organizer* class Performs organization on cepton unorganized points. Creates an angular grid, places each point within that grid and outputs a point for each location in the grid in a row/col format. Thread safe. Defaults to a 0.4deg spaced grid.

Public Types

enum OrganizerMode
Values:

RECENT
Output the most recent point from the frame that fell within the grid

CENTER
Output the center of the grid. Uses median point distance.

Public Functions

Organizer(cepton_sdk::*SensorInformation* *sensor_info*)
Organizer.

Parameters

- `sensor_info`: Sensor info for organizer. Used to set min/max angles

void **organize_points**(**const** int *num_points_in*, **const** int *n_returns*, **const** CeptonSensorImagePoint ***const** *unorganized_points*, cepton_sdk::util::*OrganizedCloud* &*organized_points*)
organize_points

Parameters

- `[in] num_points_in`: Number of unorganized points
- `[in] n_returns`: Number of returns
- `[in] unorganized_points`: Unorganized points to proces
- `[out] organized_points`: Points in organized form

void **mode**(*OrganizerMode mode*)
mode

Parameters

- `mode`: Change the mode of the organizer. [RECENT] Points are the most recent which fill within the grid. [CENTER] Points outputted are at the center of the grid. More even spacing but less accurate.

void **binSize**(float *bin_size*)
binSize Change the bin size of the organizer

Parameters

- `bin_size`: The horizontal and vertical bin size to set. In radians

void **settings**(*OrganizerSettings organizer_settings*)
settings

Parameters

- `organizer_settings`: Change organizer settings

Organizer::*OrganizerSettings* **settings**()
settings

Return The settings the organizer is using

struct OrganizerSettings

Public Members

float **horizontal_range_radians** = to_radians(70.f)

float **vertical_range_radians** = to_radians(30.f)

float **horizontal_bin_size_radians** = to_radians(0.4f)

float **vertical_bin_size_radians** = to_radians(0.4f)

OrganizerMode **mode** = *OrganizerMode*::RECENT

CHAPTER

TEN

SAMPLES

10.1 Basic

Listing 1: samples/basic.cpp

```cpp
/**
 * Sample code for general sdk usage.
 */
#include <iostream>
#include <string>
#include <vector>

#include <cepton_sdk_api.hpp>

class FramesListener {
 public:
  void on_image_frame(cepton_sdk::SensorHandle handle, std::size_t n_points,
                      const cepton_sdk::SensorImagePoint *c_image_points) {
    // Get sensor info
    cepton_sdk::SensorInformation sensor_info;
    cepton_sdk::api::check_error(
        cepton_sdk::get_sensor_information(handle, sensor_info));

    // Print info
    if (i_frame < 5) {
      std::printf("Received %i points from sensor %i\n", (int)n_points,
                  (int)sensor_info.serial_number);
    }
    ++i_frame;
  }

 private:
  std::size_t i_frame = 0;
};

int main(int argc, char **argv) {
  std::string capture_path;
  if (argc >= 2) capture_path = argv[1];

  // Initialize
  auto options = cepton_sdk::create_options();
  cepton_sdk::api::check_error(
      cepton_sdk::api::initialize(options, capture_path));
```

(continues on next page)

(continued from previous page)

```

  // Get sensor
  std::printf("Waiting for sensor to connect...\n");
  while (cepton_sdk::get_n_sensors() == 0)
    cepton_sdk::api::check_error(cepton_sdk::api::wait(0.1f));
  cepton_sdk::SensorInformation sensor_info;
  cepton_sdk::api::check_error(
      cepton_sdk::get_sensor_information_by_index(0, sensor_info));
  std::printf("Sensor: %d\n", (int)sensor_info.serial_number);

  // Listen for frames
  std::printf("Listening for frames...\n");
  cepton_sdk::api::SensorImageFrameCallback callback;
  cepton_sdk::api::check_error(callback.initialize());
  FramesListener frames_listener;
  callback.listen(&frames_listener, &FramesListener::on_image_frame);

  // Run
  cepton_sdk::api::check_error(cepton_sdk::api::wait());

  // Deinitialize
  cepton_sdk::api::check_error(cepton_sdk::deinitialize());
}
```

10.2 C Basic

Listing 2: samples/c_basic.c

```
/**
 * Sample code for general C sdk usage.
 */
#include <stdio.h>
#include <time.h>

#include <cepton_sdk.h>

int frames_got = 0;
time_t first_frame_time = 0;
time_t current_frame_time = 0;

void image_frame_callback(CeptonSensorHandle handle, size_t n_points,
                          const struct CeptonSensorImagePoint *c_points,
                          void *user_data) {
  time_t t = time(NULL);

  if (frames_got == 0) first_frame_time = t;
  frames_got++;
  if (frames_got < 50)
    printf("Frame: %4d Time: %lld\n", frames_got, (long long)t);
  else
    printf("Frame: %4d Time: %lld Frame rate: %.1fHz\n", frames_got,
           (long long)t, frames_got * 1.0 / (t - first_frame_time));
}

```

(continues on next page)

(continued from previous page)

```
int main() {
  printf("Start\n");
  fflush(stdout);
  struct CeptonSDKOptions options = cepton_sdk_create_options();
  options.frame.mode = CEPTON_SDK_FRAME_TIMED;
  options.frame.length = 0.1f;
  CeptonSensorErrorCode ret;
  ret = cepton_sdk_initialize(CEPTON_SDK_VERSION, &options, NULL, NULL);
  if (ret != CEPTON_SUCCESS) printf("%s\n", cepton_get_error_code_name(ret));
  ret = cepton_sdk_listen_image_frames(image_frame_callback, NULL);
  if (ret != CEPTON_SUCCESS) printf("%s\n", cepton_get_error_code_name(ret));

  while (frames_got < 100)
    ;  // Just spin loop for lack of cross platform sleep in C

  cepton_sdk_deinitialize();

  return 0;
}
```

10.3 Callback

Listing 3: samples/callback.cpp

```
/**
 * Sample code for callback usage.
 */
#include <cepton_sdk_api.hpp>

// Sample global callback.
void on_image_frame(cepton_sdk::SensorHandle handle, std::size_t n_points,
                    const cepton_sdk::SensorImagePoint *c_image_points) {}

// Sample member callback.
class FramesListener {
 public:
  void on_image_frame(cepton_sdk::SensorHandle handle, std::size_t n_points,
                      const cepton_sdk::SensorImagePoint *c_image_points) {}
};

int main(int argc, char **argv) {
  // Initialize
  cepton_sdk::api::check_error(cepton_sdk::api::initialize());
  cepton_sdk::api::SensorImageFrameCallback callback;
  cepton_sdk::api::check_error(callback.initialize());

  // Listen lambda
  callback.listen([](cepton_sdk::SensorHandle handle, std::size_t n_points,
                     const cepton_sdk::SensorImagePoint *c_image_points) {});

  // Listen global function
  callback.listen(on_image_frame);

  // Listen member function
```

(continues on next page)

(continued from previous page)

```
  FramesListener frames_listener;
  callback.listen(&frames_listener, &FramesListener::on_image_frame);
}
```

10.4 Error

Listing 4: samples/error.cpp

```
/**
 * Sample code for error callback usage.
 */
#include <cepton_sdk_api.hpp>

int main(int argc, char** argv) {
  // Initialize
  cepton_sdk::api::SensorErrorCallback error_callback;
  error_callback.listen(
      [&](cepton_sdk::SensorHandle handle,
          const cepton_sdk::SensorError& error) { throw error; });
  cepton_sdk::api::check_error(cepton_sdk::initialize(
      CEPTON_SDK_VERSION, cepton_sdk::create_options(),
      error_callback.global_on_callback, &error_callback));
}
```

10.5 Sensor

Listing 5: samples/sensor.cpp

```
/**
 * Sample code for sensor information.
 */
#include <string>

#include <cepton_sdk_api.hpp>

int main(int argc, char **argv) {
  std::string capture_path;
  if (argc >= 2) capture_path = argv[1];

  // Initialize
  auto options = cepton_sdk::create_options();
  cepton_sdk::api::check_error(
      cepton_sdk::api::initialize(options, capture_path));
  cepton_sdk::api::check_error(cepton_sdk::api::wait(5.0f));

  // Get all sensors
  const int n_sensors = (int)cepton_sdk::get_n_sensors();
  for (int i = 0; i < n_sensors; ++i) {
    cepton_sdk::SensorInformation sensor_info;
    cepton_sdk::api::check_error(
        cepton_sdk::get_sensor_information_by_index(i, sensor_info));
```

(continues on next page)

(continued from previous page)

```
    std::printf("%i: %s\n", (int)sensor_info.serial_number,
                sensor_info.model_name);
  }
}
```

10.6 Advanced

10.6.1 Frame

Listing 6: samples/advanced/frame.cpp

```
/**
 * Sample code for custom frame accumulation.
 */
#include <cepton_sdk_api.hpp>

int main(int argc, char **argv) {
  std::string capture_path;
  if (argc >= 2) capture_path = argv[1];

  // Initialize
  auto options = cepton_sdk::create_options();
  cepton_sdk::api::check_error(
      cepton_sdk::api::initialize(options, capture_path));
  cepton_sdk::api::SensorImageFrameCallback callback;
  cepton_sdk::api::check_error(callback.initialize());

  // Get sensor
  while (cepton_sdk::get_n_sensors() == 0)
    cepton_sdk::api::check_error(cepton_sdk::api::wait(0.1f));
  cepton_sdk::SensorInformation sensor_info;
  cepton_sdk::api::check_error(
      cepton_sdk::get_sensor_information_by_index(0, sensor_info));

  // Create accumulator
  auto frame_options = cepton_sdk::create_frame_options();
  frame_options.mode = CEPTON_SDK_FRAME_TIMED;
  frame_options.length = 0.1f;
  cepton_sdk::util::FrameAccumulator accumulator(sensor_info);
  cepton_sdk::api::check_error(accumulator.set_options(frame_options));
  callback.listen(
      [&](cepton_sdk::SensorHandle handle, std::size_t n_points,
          const cepton_sdk::SensorImagePoint *const c_image_points) {
        if (handle != sensor_info.handle) return;
        accumulator.add_points((int)n_points, c_image_points);
      });

  // Listen
  accumulator.callback.listen(
      [&](int n_points,
          const cepton_sdk::SensorImagePoint *const c_image_points) {
        std::printf("Received %i points\n", n_points);
      });
```

(continues on next page)

(continued from previous page)

```
43
44   cepton_sdk::api::check_error(cepton_sdk::api::wait(5.0f));
45 }
```

10.6.2 Network

Listing 7: samples/advanced/network.cpp

```
1  /**
2   * Sample code for custom networking.
3   */
4  #include <asio.hpp>
5
6  #include <cepton_sdk_api.hpp>
7
8  using asio::ip::udp;
9
10 class SocketListener {
11  public:
12   SocketListener() : m_socket(m_io_service, udp::v4()) {
13     m_socket.set_option(asio::socket_base::reuse_address(true));
14     m_socket.bind(udp::endpoint(udp::v4(), 8808));
15   }
16
17   void run() {
18     listen();
19     m_io_service.run_for(std::chrono::seconds(5));
20   }
21
22   void listen() {
23     m_socket.async_receive_from(
24         asio::buffer(m_buffer), m_end_point,
25         [this](const asio::error_code& error, std::size_t buffer_size) {
26           if (buffer_size == 0) return;
27           if (error == asio::error::operation_aborted) return;
28           const CeptonSensorHandle handle =
29               m_end_point.address().to_v4().to_ulong();
30           // For more accurate timestamps, a separate network receive thread
31           // should be
32           // used.
33           const int64_t timestamp = cepton_sdk::util::get_timestamp_usec();
34           cepton_sdk::api::check_error(cepton_sdk::mock_network_receive(
35               handle, timestamp, m_buffer.data(), buffer_size));
36           listen();
37         });
38   }
39
40  private:
41   asio::io_context m_io_service;
42   udp::socket m_socket;
43   udp::endpoint m_end_point;
44   std::array<uint8_t, 4096> m_buffer;
45 };
46
47 int main() {
```

(continues on next page)

(continued from previous page)

```
  // Initialize sdk
  auto options = cepton_sdk::create_options();
  options.control_flags |= CEPTON_SDK_CONTROL_DISABLE_NETWORK;
  options.frame.mode = CEPTON_SDK_FRAME_COVER;
  cepton_sdk::api::check_error(cepton_sdk::api::initialize(options));

  // Listen for points
  cepton_sdk::api::SensorImageFrameCallback callback;
  cepton_sdk::api::check_error(callback.initialize());
  callback.listen([](cepton_sdk::SensorHandle handle, std::size_t n_points,
                     const cepton_sdk::SensorImagePoint* c_image_points) {
    std::printf("Received %i points from sensor %lli\n", (int)n_points,
                (long long)handle);
  });

  SocketListener listener;
  listener.run();
}
```

10.6.3 Process Multi

Listing 8: samples/advanced/process_multi.cpp

```
/**
 * Sample code for processing multiple sensor data.
 */
#include <string>
#include <vector>

#include <cepton_sdk_api.hpp>

struct Frame {
  int64_t timestamp;
  std::map<cepton_sdk::SensorHandle, std::vector<cepton_sdk::SensorImagePoint>>
      image_points_dict;
};

class FrameAccumulator {
 public:
  void on_image_frame(
      cepton_sdk::SensorHandle handle, std::size_t n_points,
      const cepton_sdk::SensorImagePoint* const c_image_points) {
    cepton_sdk::util::LockGuard lock(m_mutex);

    auto& image_points = m_image_points_dict[handle];
    image_points.reserve(image_points.size() + n_points);
    image_points.insert(image_points.end(), c_image_points,
                        c_image_points + n_points);

    check_and_publish();
  }

 private:
  void check_and_publish() {
    const auto timestamp = cepton_sdk::api::get_time();
```

(continues on next page)

(continued from previous page)

```
    if (timestamp < m_timestamp) return;
    if ((timestamp - m_timestamp) < int64_t(m_frame_length * 1e6f)) return;
    m_timestamp = timestamp;

    auto frame = std::make_shared<Frame>();
    frame->timestamp = timestamp;
    frame->image_points_dict = m_image_points_dict;
    m_image_points_dict.clear();
    queue.push(frame);
  }

 public:
  cepton_sdk::util::SingleConsumerQueue<Frame> queue;

 private:
  std::timed_mutex m_mutex;
  float m_frame_length = 0.1f;
  int64_t m_timestamp = 0;
  std::map<cepton_sdk::SensorHandle, std::vector<cepton_sdk::SensorImagePoint>>
      m_image_points_dict;
};

int main(int argc, char** argv) {
  std::string capture_path;
  if (argc >= 2) capture_path = argv[1];

  auto options = cepton_sdk::create_options();
  cepton_sdk::api::check_error(
      cepton_sdk::api::initialize(options, capture_path));
  cepton_sdk::api::SensorImageFrameCallback callback;
  cepton_sdk::api::check_error(callback.initialize());
  if (cepton_sdk::capture_replay::is_open())
    cepton_sdk::api::check_error(cepton_sdk::capture_replay::resume());

  FrameAccumulator accumulator;
  callback.listen(&accumulator, &FrameAccumulator::on_image_frame);

  while (true) {
    const auto frame = accumulator.queue.pop(0.01f);
    if (!frame) continue;
    // Do processing
  }
}
```

10.6.4 Process Single

Listing 9: samples/advanced/process_single.cpp

```
/**
 * Sample code for processing single sensor data.
 */
#include <memory>
#include <vector>

```

(continues on next page)

(continued from previous page)

```cpp
#include <cepton_sdk_api.hpp>

struct Frame {
  int64_t timestamp;
  cepton_sdk::SensorHandle handle;
  std::vector<cepton_sdk::SensorImagePoint> image_points;
};

int main(int argc, char **argv) {
  std::string capture_path;
  if (argc >= 2) capture_path = argv[1];

  auto options = cepton_sdk::create_options();
  cepton_sdk::api::check_error(
      cepton_sdk::api::initialize(options, capture_path));
  cepton_sdk::api::SensorImageFrameCallback callback;
  cepton_sdk::api::check_error(callback.initialize());
  if (cepton_sdk::capture_replay::is_open())
    cepton_sdk::api::check_error(cepton_sdk::capture_replay::resume());

  cepton_sdk::util::SingleConsumerQueue<Frame> queue;
  callback.listen([&](cepton_sdk::SensorHandle handle, std::size_t n_points,
                      const cepton_sdk::SensorImagePoint *c_image_points) {
    auto frame = std::make_shared<Frame>();
    frame->timestamp = cepton_sdk::api::get_time();
    frame->handle = handle;
    frame->image_points.reserve(n_points);
    frame->image_points.insert(frame->image_points.end(), c_image_points,
                               c_image_points + n_points);
    queue.push(frame);
  });

  while (true) {
    const auto frame = queue.pop(0.01f);
    if (!frame) continue;
    // Do processing
  }
}
```

10.6.5 Replay

Listing 10: samples/advanced/replay.cpp

```cpp
/**
 * Sample code for custom packet replaying.
 */
#include <cepton_sdk/capture.hpp>
#include <cepton_sdk_api.hpp>

class CaptureReplay {
 public:
  CaptureReplay(const std::string& path) {
    cepton_sdk::api::check_error(m_capture.open_for_read(path));

    cepton_sdk::api::check_error(
```

(continues on next page)

(continued from previous page)

```
        cepton_sdk_set_control_flags(CEPTON_SDK_CONTROL_DISABLE_NETWORK,
                                     CEPTON_SDK_CONTROL_DISABLE_NETWORK));
    cepton_sdk::api::check_error(cepton_sdk_clear());
  }

  ~CaptureReplay() {
    m_capture.close();
    if (cepton_sdk_is_initialized()) {
      cepton_sdk::api::check_error(cepton_sdk_clear());
    }
  }

  void run() {
    while (true) {
      cepton_sdk::Capture::PacketHeader header;
      const uint8_t* data;
      cepton_sdk::api::check_error(m_capture.next_packet(header, data));

      const cepton_sdk::SensorHandle handle =
          (cepton_sdk::SensorHandle)header.ip_v4 |
          CEPTON_SENSOR_HANDLE_FLAG_MOCK;
      cepton_sdk::api::check_error(cepton_sdk_mock_network_receive(
          handle, header.timestamp, data, header.data_size));
    }
  }

 private:
  cepton_sdk::Capture m_capture;
};

int main(int argc, char** argv) {
  if (argc < 2) return -1;
  const std::string capture_path = argv[1];

  // Initialize sdk
  auto options = cepton_sdk::create_options();
  options.control_flags |= CEPTON_SDK_CONTROL_DISABLE_NETWORK;
  options.frame.mode = CEPTON_SDK_FRAME_COVER;
  cepton_sdk::api::check_error(cepton_sdk::api::initialize(options));

  // Listen for points
  cepton_sdk::api::SensorImageFrameCallback callback;
  cepton_sdk::api::check_error(callback.initialize());
  callback.listen([](cepton_sdk::SensorHandle handle, std::size_t n_points,
                     const cepton_sdk::SensorImagePoint* c_image_points) {
    std::printf("Received %i points from sensor %lli\n", (int)n_points,
                (long long)handle);
  });

  CaptureReplay replay(capture_path);
  replay.run();
}
```

10.6.6 Stray

Listing 11: samples/advanced/stray.cpp

```cpp
/**
 * Sample code for stray filter usage.
 */
#include <vector>

#include <cepton_sdk_api.hpp>

int main(int argc, char **argv) {
  std::string capture_path;
  if (argc >= 2) capture_path = argv[1];

  // Initialize
  auto options = cepton_sdk::create_options();
  cepton_sdk::api::check_error(
      cepton_sdk::api::initialize(options, capture_path));

  cepton_sdk::api::SensorImageFrameCallback callback;
  cepton_sdk::api::check_error(callback.initialize());
  cepton_sdk::util::StrayFilter filter;
  callback.listen([&](cepton_sdk::SensorHandle handle, std::size_t n_points,
                      const cepton_sdk::SensorImagePoint *c_image_points) {
    // Get sensor
    cepton_sdk::SensorInformation sensor_info;
    cepton_sdk::api::check_error(
        cepton_sdk::get_sensor_information(handle, sensor_info));

    // Copy points to buffer
    std::vector<cepton_sdk::SensorImagePoint> image_points;
    image_points.insert(image_points.begin(), c_image_points,
                        c_image_points + n_points);

    // Filter stray
    filter.init(sensor_info);
    filter.run((int)n_points, image_points.data());
  });

  // Run
  cepton_sdk::api::check_error(cepton_sdk::api::wait(5.0f));
}
```

10.6.7 Transform

Listing 12: samples/advanced/transform.cpp

```cpp
/**
 * Sample code for transforming 3d points.
 */
#include "cepton_sdk_api.hpp"

int main() {
  // Create transform
  std::array<float, 3> translation = {0.0f, 0.0f, 1.0f};
  std::array<float, 4> rotation = {1.0f, 0.0f, 0.0f, 0.0f};
```

(continues on next page)

(continued from previous page)

```
  auto compiled_transform = cepton_sdk::util::CompiledTransform::create(
      translation.data(), rotation.data());

  // Apply transform
  cepton_sdk::util::SensorPoint point = {};
  compiled_transform.apply(point.x, point.y, point.z);
  printf("Point: [%f, %f, %f]\n", point.x, point.y, point.z);
}
```

INDEX

C